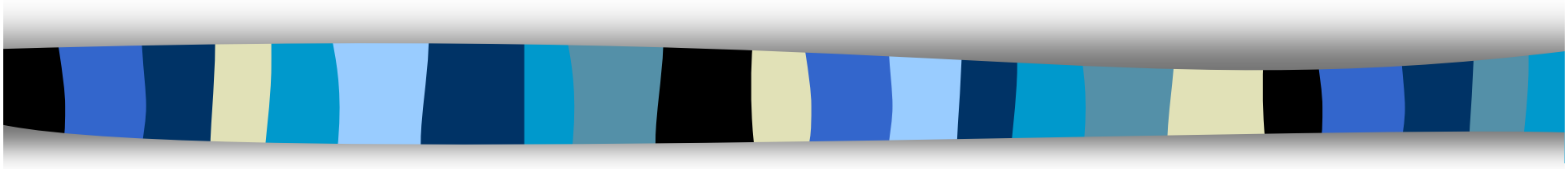
Recursão



Introdução à Ciência da Computação II (2009/2010)

Rosane Minghim

Apoio na confecção: Rogério Eduardo Garcia

Danilo Medeiros Eler

Definição

- Um objeto é dito ser recursivo se ele é definido parcialmente em termos de si próprio.
- Recursão é uma técnica poderosa em definições matemáticas. O poder da recursão está na possibilidade de se definir elementos com base em versões mais simples deles mesmos.
- Ex: potência positiva (n>=0) de um número X

  $x^n = 1$ se n=0

  $x^n = x * x^{n-1}$ se n>0

Recursividade

- Algoritmos recursivos são principalmente apropriados quando o problema a ser resolvido, a função a ser computada, ou os dados já estão definidos em termos recursivos ou por indução.
- Mas não significa que tal definição recursiva garanta a melhor solução do problema em termos de eficiência.
- A única ferramenta necessária para expressar operações recursivamente é o próprio procedimento ou a função, que tem a capacidade de invocar a si próprio.

Características

- Uma condição de parada, isto é, algum evento que encerre a auto-chamada consecutiva. No caso do fatorial, isso ocorre quando a função é chamada com parâmetro (n) igual a 1. Um algoritmo recursivo precisa garantir que esta condição será alcançada.

- Uma mudança de "estado" a cada chamada, isto é, alguma "diferença" entre uma chamada e a próxima. No caso do fatorial, o parâmetro n é decrementado a cada chamada.

Padrão de definições recursivas

a) o caso simples é definido explicitamente
   - Ex1: 0! = 1
     Ex2: a*1 = a

b) outros casos são definidos aplicando-se alguma operação que inclua o caso simples na sequência de operações necessária para resolvê-la.
   - Ex1: n! = n * (n-1)!
     Ex2: a* b = a * (b-1) + a

Obs. Nos casos acima a definição recursiva funciona para $n \geq 0$ e $b > 0$

Exemplo – Fatorial

Subprograma fatorial(n):inteiro

e: n: inteiro {número do qual seria calculado o Fatorial}

r: inteiro fatorial de n

pré-condição: n>0

```
início
  se n = 1 então
    retorne (1)
  senão
    retorne (n*fatorial(n-1))
  fim se
fim
```

Percorrendo o Algoritmo

algoritmo irá acionar o subprograma fatorial através do seguinte comando:

escreva('fatorial de 5 = ',fatorial(5))

A sequência de chamadas ao subprograma é dada por:

```
fatorial(5)
  {n=5}
  fatorial(4)
    {n=4}
    fatorial(3)
      {n=3}
      fatorial(2)
        {n=2}
        fatorial(1)
          {n=1}
            fatorial(0)
              retorna (1)
          retorna(1*1)
        retorna (2*1)
      retorna(3*2)
    retorna(4*6)
  retorna(5*24)
```

Saída impressa: fatorial de 5 = 120

Exemplo

Função recursiva para encontrar a soma dos elementos de um vetor a.

- Se definirmos s(k) como a soma dos valores de v com índices de 1 a k, podemos escrever:

  1) s(k) = s(k-1) + v[k], 1 <= k <= n

  2) s(0) = 0

```
Subprograma soma_vet(v,n):real
e: v:vetor_real
n:inteiro
r: soma dos elementos do vetor v
início
   Se n=0 então
      retorne 0
   senão
      retorne (v[n] + soma(v,n-1))
   fim se
fim
```

Exemplo: Busca Binária

```
Subprograma buscabin(v,i,j,x,pos)
e:v:vetor vetor de elementos simples
x:elemento elemento a ser procurado no vetor
i:índice inferior vetor
j:índice superior do vetor
s: pos: {posição onde o elemento foi encontrado (ou -1 se não for)}
variável
   med :inteiro;
início
      Se i<=j então
        med ← quociente((i+j),2)
       se v[med]=x então
         pos ←med
       senão
          se v[med] < x então
            buscabin(v,med+1,j,x,pos)
          senão
              buscabin(v,i,med-1,x,pos)
          fim se
```

continuação

```
          fim se
       senão
          pos ← -1
       fim se
fim
```

Recursão de Cauda

```
1 Subprograma impvet(v,i,n)
2
3 e: v:vetor
4 i:inteiro posi»c~ao atual dentro do vetor
5 n:inteironúmero de elementos do vetor
6
7 {subprograma recursivo para impressão do
8 vetor}
9
10 início
11     Se i <= n então
12       escreva(vet[i])
13       impvet(v,i+1,n)
14     fim se
15 fim
```

```
1 Subprograma impvet(v,i,n)
2
3 e: v:vetor
4 i:inteiro posi»c~ao atual dentro do vetor
5 n:inteironúmero de elementos do vetor
6
7 {subprograma recursivo para impressão do
8 vetor}
9
10 início
11     Se i <= n então
12       impvet(v,i+1,n)
13       escreva(vet[i])
14     fim se
15 fim
```

Qual a diferença entre os dois subprogramas?

Recursão de Cauda

- Os subprogramas acima têm definições que seriam idênticas não fosse pelas linhas 12 e 13, que estão invertidas.
- No primeiro algoritmo, o i-ésimo valor é escrito e então o subprograma é chamado para os elementos subsequentes do vetor.
- No segundo algoritmo a chamada ocorre antes da impressão. Como consequência, o primeiro algoritmo tem como resultado a impressão dos elementos do vetor na ordem em que se encontram armzenados no vetor, e o segundo imprime os valores na ordem inversa, isto é, o primeiro valor a ser impresso é v[n].
- A recursão do primeiro algoritmo é chamada de Recursão de Cauda. A soma de vetores e o fatorial também são uma recursão de cauda. Já a busca binária e a permutação não são.

Recursão de Cauda

- Todo algoritmo recursivo pode ser transformado em um interativo.
- Muitas vezes isso não é uma tarefa fácil, principalmente quando a natureza do problema é muito mais facilmente (e claramente) expressa de forma recursiva.
- A recursão de cauda, no entanto, é uma estrutura das mais fáceis de se traduzir para uma versão iterativa.

Recursão de Cauda

- O formato de uma função recursiva de cauda P pode sempre ser mapeado para:

  a) P ≡ if (condição) then {S;P}.

  b) P ≡ S; if condição then P.

  (veja que isso ocorre nos exemplos dados).

- Iterativamente esses formatos são equivalentes a:

  a) C; while (condição) then {S;C1}.

  b) C; repita {S;C1} until (not condição)

  Once C é via de regra um comando de inicialização e C1 um comando de atualização

Recursão de Cauda – Exemplos de conversão

```
1 Subprograma impvet(v,i,n)
2
3 e: v:vetor
4 i:inteiro posição atual dentro do vetor
5 n:inteironúmero de elementos do vetor
6
7 {subprograma recursivo para impressão do
8 vetor – imprime elementos de i a n}
9
10 início
11     Se i <= n então
12       escreva(vet[i])
13       impvet(v,i+1,n)
14     fim se
15 fim
```

```
1 Subprograma impvet(v,i,n)
2
3 e: v:vetor
4 i:inteiro posição atual dentro do vetor
5 n:inteironúmero de elementos do vetor
6
7 {subprograma recursivo para impressão do
8 vetor – imprime elementos de i a n}
9
10 início
11     j <- i
11     enquanto j <= n then
12       escreva(vet[j])
13       j <- j+1
14     fim enquanto
15 fim
```

Exemplo de conversão

```
Subprograma fatorial(n):inteiro

e: n: inteiro
{número do qual seria calculado o Fatorial}

r: inteiro fatorial de n

pré-condição: n>0

início
  se n > 1 então
     retorne (n*fatorial(n-1))
   senão
      retorne (1)
  fim se
fim
```

```
Subprograma fatorial(n):inteiro

e: n: inteiro
{número do qual seria calculado o Fatorial}

r: inteiro fatorial de n

pré-condição: n>0

início
  ini <- n
  fat <- 1
  enquanto ini > 1
     fat <- fat * ini
     ini <- ini -1
  fim enquanto
fim
```

Recursão Indireta

Dados dois subprogramas ***a*** e ***b***, uma recusão indireta é quando ***a*** chama ***b*** que chama ***a***.

Exemplo:

Definição de número par: é um número divisível por dois.

Definição de um número ímpar: é um número não divisível por dois.

Definição recursiva de número par: 0 é par. n é par se n-1 é ímpar, n>0.

Definição recursiva de número ímpar: 1 é ímpar. n é ímpar se n-1 é par.

Recursão Indireta

```
Subprograma par(x):lógico
e:x:inteiro
r:verdadeiro se x é par, falso caso contrário
início
    Se x=0 então
       retorne(verdadeiro)
    senão
        Se x=1 então
           retorne(falso)
        senão
          retorne (impar(x-1))
        fim se
     fim se
fim
```

```
Subprograma ímpar(x):lógico
e:x:inteiro
r:verdadeiro se x é ímpar, falso caso
contrário
início
      Se x=0 então
        retorne(falso)
      senão
        Se x=1 então
          retorne(verdadeiro)
        senão
          retorne (par(x-1))
        fim se
      fim se
fim
```

Dados dois subprogramas ***a*** e ***b***, uma recusão indireta é quando ***a*** chama ***b*** que chama ***a***.

Esquemas não apropriados à recursão

1. Quando existe uma única chamada do procedimento recursivo no fim ou no começo da rotina, o procedimento é facilmente tranformado numa iteração simples. É boa politica não usar recursão quando existe um algoritmo iterativo igualmente claro que resolva o problema. É o caso do fatorial e da soma de vetores vistos anteriormente.
2. Quando o uso de recursão acarreta num número maior de cálculos

Exemplo– Sequência de Fibonacci

- A sequência de Fibonacci é uma série de número inteiros. O n-ésimo elemento da série é dado por:

$$fibonacci(n) = \begin{cases} 0 & se \quad n = 0 \\ 1 & se \quad n = 1 \\ fibonacci(n-1) + fibonacci(n-2) & se \quad n > 1 \end{cases}$$

- Apesar de ser evidentemente recursiva pode-se perceber que, para calcular um dado elemento da sequência, os dois termos da soma (no caso n>1), cálculos serão repetidos.

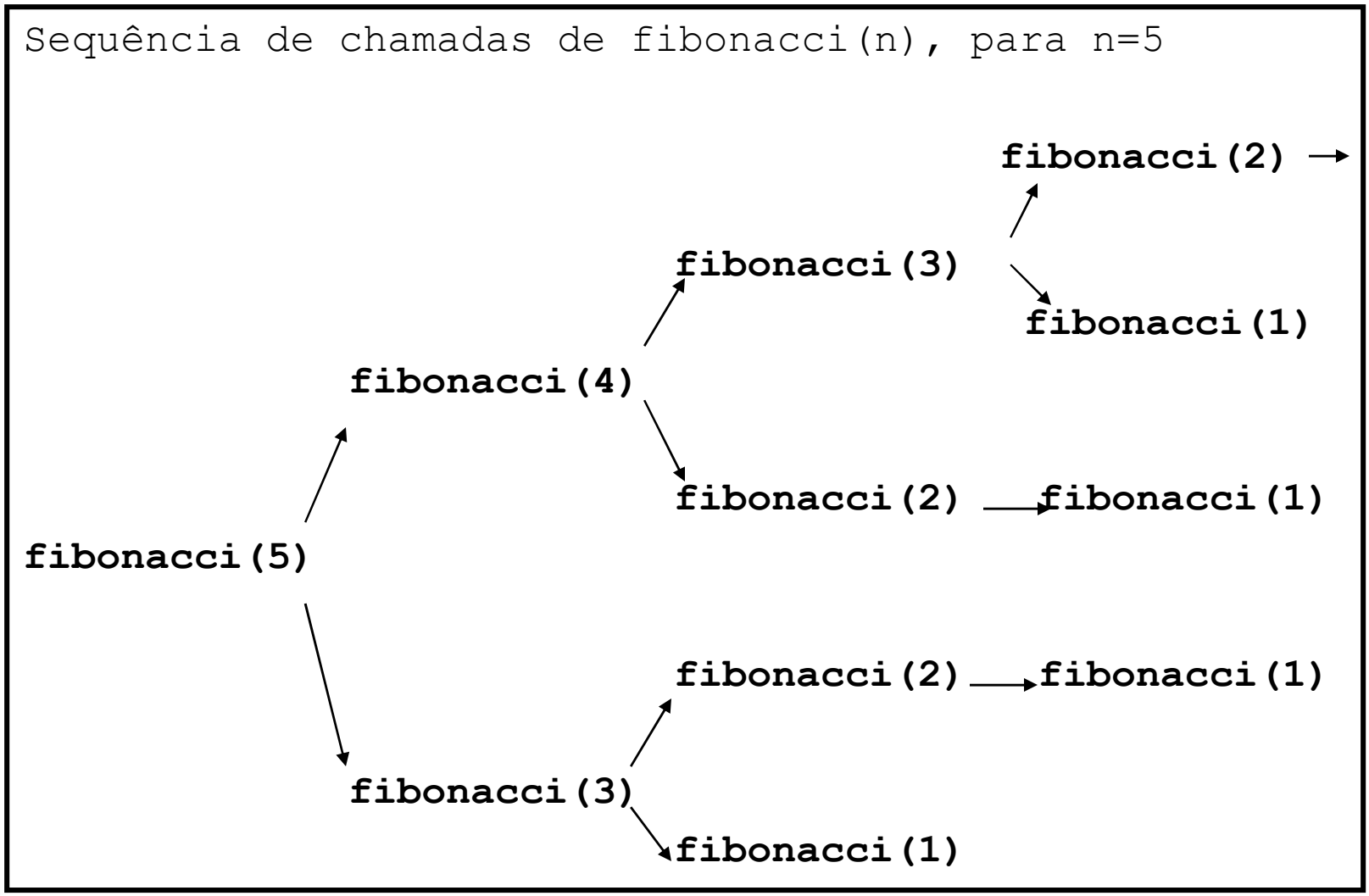
- Sequência de Fibonacci: 0, 1, 1, 2, 3, 5, 8,…

Fibonacci – versão recursiva

```
Subprograma fibonacci(n): integer
e:n: índice do elemento da série
r: n-ésimo número de Fibonacci (n>=0)

início
  se n=0 então
    retorne (0)
  senão
    se n = 1 então
       retorne(1)
    senão
       retorne(fibonacci(n-1)+fibonacci(n-2))
    fim se
  fim se
fim
```

Fibonacci – versão recursiva



Fica pior: e se quiséssemos imprimir a sequência de 1 a 5? a 10? a 100?

Fibonacci – versão iterativa

```
Subprograma fibonacci(n): integer
e:n: índice do elemento da série
r: n-ésimo número de Fibonacci (n>=0)
início
  se n=0 então
    fib <- 0
  senão
    se n = 1 então
       fib <- 1
    senão
       n_2 <- 0
       n_1 <- 1
       fib <- 1
       para i de 3 até n faça
          fib <- fib + n_1 + n_2
          n_2 <- n_1
          n_1 <- fib
       fim para
    fim se
  fim se
  retorne(fib)
fim
```

Use Recursão Quando

- *O problema é naturalmente recursivo (clareza) e a versão recursiva do algoritmo não gera ineficiência evidente se comparado com a versão iterativa do mesmo algoritmo.*
- *O algoritmo se torna compacto sem perda de clareza ou generalidade.*
- *É possível prever que o número de chamadas ou a carga sobre a pilha de passagem de parâmetros não irá causar interrupção do processo.*

Não Use Recursão Quando

- *A solução recursiva causa ineficiência se comparada com uma versão iterativa.*
- *A recursão é de cauda.*
- *Parâmetros consideravelmente grandes têm que ser passados por valor.*
- *Não é possível prever se o número de chamadas recursivas irá causar sobrecarga da pilha de passagem de parâmetros.*

Recursão em Pascal

- Em Pascal a recursão é implementada, do ponto de vista sintático, da mesma forma que em pseudo-código, ou seja, através de procedimentos e funções que chamam a si próprios direta ou indiretamente.
- Do ponto de vista de implementação as considerações sobre chamadas de subprogramas e passagem de parâmetros valem, isto é, cada chamada a um subprograma recursivo gera uma nova alocação de parâmetros e variáveis locais e uma chamada independente da anterior.
- Múltiplas chamadas sobrecarregam a pilha de passagem de parâmetros e variáveis locais dependendo da “profundidade” da recursão.

Recursão em Pascal - Exemplos

```
type
   vetor = array [1..max] of integer;


Procedure impvet(var v:vetor; i,n: integer);
{ e: v:vetor
 i:inteiro posição atual dentro do vetor
 n:inteironúmero de elementos do vetor
 subprograma recursivo para impressão do
 v etorv  – imprime elementos de i a n, em
       ordem invertida}
begin
    if  i <= n then
    begin
       impvet(v,i+1,n);
       writeln(vet[i]);
    end
end;
```

Recursão Indireta em Pascal

- Em Pascal a recursão indireta apresenta o seguinte problema:
  - Quando uma função chama a outra, é necessário que a função chamada tenha sido definida previamente.
  - Em Recursões indiretas, uma das funções chama uma função que ainda não foi definida (ex. par/ímpar).

- Para solucionar este problema, existe o recurso ‘forward’ do Pascal que permite definir o cabeçalho de uma função, mas `adiar' a apresentação do seu código interno para um outro trecho do mesmo arquivo.

- A implementação das funções par / ímpar em Pascal é dada a seguir.

Recursão em Pascal – Recursão Indireta - Exemplo

```
Function impar(x:integer):boolean; forward;
```

```
Function par(x:integer):boolean;

{e:x:inteiro
 r:verdadeiro se x x par,falso caso
contrário }

begin
  if x=0 then
    par := TRUE
  else
    if x=1 then
      par := FALSE
    else
      par := impar(x-1);
end;
```

```
Function impar(x:integer):boolean;

{e:x:inteiro
 r:verdadeiro se x eh impar, falso caso
contrario }

begin
  if x=0 then
    impar:=FALSE
  else
    if x=1 then
      impar:=TRUE
    else
      impar:=par(x-1);
end;
```

Exemplo: permutação

Procedimento recursivo para listar todas as permutações de N elementos distintos de uma cadeia de caracteres s[1] ... s[n]. Número de permutações = n!

- Processo:

  Se n>1 mantenha s[n] em seu lugar e gere todas as permutações de s[1] ...s[n-1] chamando permute(n-1,s) caso contrário imprima a cadeia.

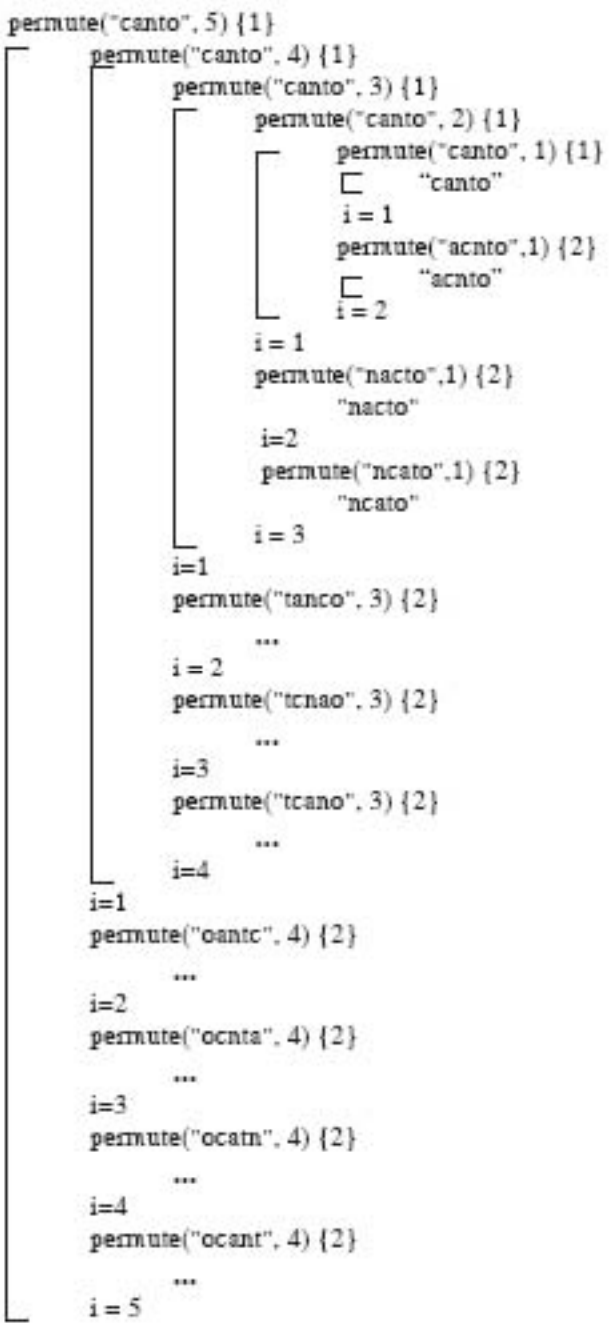
  Troque s[n] com s[i], e repita o processo acima. Faça isso para i = 1,…,n-1.

Exemplo

```
subprograma permute(s,n)
e:s: cadeia a ser permutada
  n: índice do final da sub-cadeia a ser permutada
variáveis
  i:inteiro
início
 se n = 1 então
   escreva(s)
 senão
   permute(s,n-1) {1}
 fim se
 para i de 1 até n-1 faça
   troque s[n] com s[i]
   permute(s,n-1)  {2}
 fim para
fim
```

Exemplo

percorrendo o procedimento permute



Permutação em Pascal

```
Procedure permute(cad:cadeia;n:integer);
{e: cadeia: a cadeia decaracteres
 n:inteiro: o numero de elementos da sub-cadeia a ser     permutada}
{Este supprograma imprime todas as permutações dos n primeiroscaracteres da cadeia
        cad}
var
 i:integer;
 aux:char;
begin
if n=1 then
   writeln(cad)
else
begin
 permute(cad,n-1);
 for i:=1 to n-1 do
 begin
 {troque o caracter da posicao n com um outro da cadeia}
    aux := cad[i];
    cad[i] := cad[n];
    cad[n] := aux;
    {fixe o caracter da posicao n e permute a sub-cadeia  de n-1 elementos}
    permute(cad,n-1);
  end;
end;
```